1. **What are design patterns, and why are they useful?**
    - **Answer:** Design patterns are reusable solutions to common software design problems. They provide a template for how to solve a problem in a way that has been proven effective. Patterns help developers communicate more efficiently, improve code reusability, and make systems more maintainable by providing standardized approaches to software design.
2. **Can you explain the difference between creational, structural, and behavioral design patterns?**
    - **Answer:**
        - **Creational patterns** deal with object creation mechanisms, aiming to create objects in a manner suitable to the situation (e.g., Singleton, Factory).
        - **Structural patterns** focus on the composition of classes or objects, helping to form larger structures while keeping them flexible (e.g., Adapter, Composite).
        - **Behavioral patterns** are concerned with the interaction and responsibility of objects, focusing on communication between objects (e.g., Observer, Strategy).
3. **How do design patterns promote code reusability and maintainability?**
    - **Answer:** Design patterns promote code reusability by providing tested, proven development paradigms. This allows developers to avoid "reinventing the wheel." They enhance maintainability by promoting clear structure and separation of concerns, making it easier to update and manage code.

Creational Patterns

4. **What is the Singleton pattern? How would you implement it in Java?**
    - **Answer:** The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. In Java, it can be implemented using:

java

Copy code

```
public class Singleton {
   private static Singleton instance;

   private Singleton() { }

   public static synchronized Singleton getInstance() {
      if (instance == null) {
         instance = new Singleton();
```

```
        }
        return instance;
    }
}
```

5. **Explain the Factory Method pattern. How does it differ from the Abstract Factory pattern?**
    - **Answer:** The Factory Method pattern defines an interface for creating an object but lets subclasses alter the type of objects that will be created. It focuses on a single product.
        - **Difference from Abstract Factory:** The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It can create multiple products, whereas Factory Method typically creates one product.
6. **Can you describe the Builder pattern and provide a use case where it is beneficial?**
    - **Answer:** The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations. It's useful for creating objects with many optional parameters.

java

Copy code

```
public class User {
    private String name;
    private int age;

    private User(UserBuilder builder) {
        this.name = builder.name;
        this.age = builder.age;
    }

    public static class UserBuilder {
        private String name;
        private int age;

        public UserBuilder setName(String name) {
            this.name = name;
```

```
        return this;
    }

    public UserBuilder setAge(int age) {
        this.age = age;
        return this;
    }

    public User build() {
        return new User(this);
    }
  }
}
```

7. **What is the Prototype pattern? How do you implement it in Java?**
    - **Answer:** The Prototype pattern allows cloning of existing objects without exposing the creation logic. It involves creating a prototype interface and implementing the clone method.

java

Copy code

```
public interface Prototype {
  Prototype clone();
}

public class ConcretePrototype implements Prototype {
  private String property;

  public ConcretePrototype(String property) {
    this.property = property;
  }

  @Override
  public Prototype clone() {
```

```
        return new ConcretePrototype(property);
    }
}
```

Structural Patterns

8. **Explain the Adapter pattern and provide an example of its use in Java.**
    - **Answer:** The Adapter pattern allows incompatible interfaces to work together by creating a wrapper that translates one interface to another. For example, if you have a Duck interface and a Turkey class, an adapter can allow a turkey to behave like a duck:

java

Copy code

```
public class TurkeyAdapter implements Duck {
    private Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    @Override
    public void quack() {
        turkey.gobble();
    }
}
```

9. **What is the Decorator pattern, and how does it differ from inheritance?**
    - **Answer:** The Decorator pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class. It provides flexibility compared to inheritance, where you must create new subclasses for every combination of behaviors.

java

Copy code

```
public interface Coffee {
    double cost();
}
```

```
public class SimpleCoffee implements Coffee {
   public double cost() { return 5; }
}

public class MilkDecorator implements Coffee {
   private Coffee coffee;

   public MilkDecorator(Coffee coffee) {
      this.coffee = coffee;
   }

   public double cost() { return coffee.cost() + 1; }
}
```

10. **Can you describe the Composite pattern and give an example of where it might be used?**
    - **Answer:** The Composite pattern allows you to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions uniformly. For example, in a file system:

```
java
Copy code
public interface FileComponent {
   void showDetails();
}

public class File implements FileComponent {
   private String name;
   public File(String name) { this.name = name; }

   public void showDetails() { System.out.println(name); }
}

public class Directory implements FileComponent {
```

```
    private List<FileComponent> components = new ArrayList<>();
    private String name;

    public Directory(String name) { this.name = name; }

    public void add(FileComponent component) { components.add(component); }

    public void showDetails() {
        System.out.println(name);
        for (FileComponent component : components) {
            component.showDetails();
        }
    }
}
```

11. **What is the Facade pattern? How does it simplify interactions with complex systems?**
    - **Answer:** The Facade pattern provides a simplified interface to a complex subsystem. It hides the complexities of the system and provides an interface for the client to interact with.

```
java
Copy code
public class ComputerFacade {
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    public ComputerFacade() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    public void startComputer() {
```

```
        cpu.freeze();
        memory.load();
        hardDrive.read();
        cpu.execute();
    }
}
```

Behavioral Patterns

12. **Explain the Observer pattern. Can you provide a real-world example?**
    - **Answer:** The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. A real-world example is a weather station that notifies various display elements (like a mobile app) whenever there's a change in weather conditions.

java

Copy code

```
public interface Observer {
    void update(float temperature);
}


public class WeatherStation {
    private List<Observer> observers = new ArrayList<>();
    private float temperature;

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void setTemperature(float temperature) {
        this.temperature = temperature;
        notifyObservers();
    }

    private void notifyObservers() {
```

```
        for (Observer observer : observers) {
            observer.update(temperature);
        }
    }
}
```

13. **What is the Strategy pattern? How can it be used to implement different algorithms?**
    - **Answer:** The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the algorithm to vary independently from clients that use it. For example:

```
java
Copy code
public interface SortingStrategy {
    void sort(int[] array);
}

public class BubbleSort implements SortingStrategy {
    public void sort(int[] array) { /* sorting logic */ }
}

public class QuickSort implements SortingStrategy {
    public void sort(int[] array) { /* sorting logic */ }
}

public class Sorter {
    private SortingStrategy strategy;

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sortArray(int[] array) {
        strategy.sort(array);
```

```
    }
}
```

14. **Can you describe the Command pattern and provide an example of its application?**
    - **Answer:** The Command pattern encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations. It supports undoable operations.

java

Copy code

```
public interface Command {
    void execute();
}


public class Light {
    public void turnOn() { System.out.println("Light On"); }
    public void turnOff() { System.out.println("Light Off"); }
}


public class LightOnCommand implements Command {
    private Light light;


    public LightOnCommand(Light light) { this.light = light; }
    public void execute() { light.turnOn(); }
}
```

15. **What is the Template Method pattern? How does it promote code reuse?**
    - **Answer:** The Template Method pattern defines the skeleton of an algorithm in a base class but allows subclasses to override specific steps of the algorithm without changing its structure. It promotes code reuse by